BUGS

- **FAILURE** *(malfunzionamento)*: Risultato non corretto ottenuto dall'esecuzione di un programma. Dipende più dal comportamento del programma che dal codice *(ad esempio in assenza di informazioni su quale sia il codice)*.
- **FAULT** *(difetto)*: Un problema nel codice. Una condizione ***necessaria*** *(ma non sufficiente)* per generare un failure.
- **ERROR** *(errore)*: Ciò che causa un fault. Ad esempio l'errore dell'umano che ha implementato il codice o un problema con un documento legato ad esso.

OBBIETTIVI DEI TEST:

- Rivelare i difetti mostrando dei malfunzionamenti.
- Fornire **confidenza** della *(probabile)* correttezza del software.
  Il ***livello di confidenza*** è dato dal numero di test; più sono, più campi vengono toccati e quindi più è alto. Per fornirlo dobbiamo trovare un range di input valido che garantisca il funzionamento a livello generale.

TERMINOLOGIA

- **TEST OBLIGATIONS *(OBJECTIVE)***: Una parziale specifica di casi di test che richiede alcune proprietà ritenute importanti per il testing; sostanzialmente è l'obiettivo che si pone un insieme di test.
- **TEST CASE**: Singola prova. Un insieme di input, condizioni di esecuzione e un criterio per definire test passati/non passati.
- **TEST SUITE**: Un insieme di test cases.
- **TEST CATALOG**: Una guida per identificare le test obligations per una classe ben identificata di elementi.
- **ADEQUACY CRITERION**: Un predicato che risulta essere *True* o *False* per una coppia <*Programma, Test Suite*>. Sostanzialmente è un metodo per stabilire, data una test suite, se essa è sufficiente per testare un programma o meno.

DA DOVE VENGONO LE TEST OBLIGATION?

- **FUNZIONALI** *(Black box, basati sulle specifiche)*: Ottenute dalle specifiche del software (***conoscenza del dominio***) senza saperne l'implementazione effettiva.
  Esempio: Se una specifica richiede una qualche forma di recovery da un blackout, una test obligation dovrebbe includere la simulazione di un blackout.
- **STRUTTURALI** *(White o Glass box)*: Ottenute dal codice.
  Esempio: Eseguire ogni loop del programma una o più volte.

O ancora *(non trattati nel corso)*

- **MODEL-BASED**: Ottenute dal modello del sistema, ottenuti nel design o dal codice.
- **FAULT-BASED**: Ottenute da fault ipotizzati *(bug comuni come ad esempio il buffer overflow)*.

SELTA DEI TEST

Perché abbiamo scelto un certo test?
Sono sufficienti i vari casi di test proposti? Si/No? Perché?
Testare con 3, 5 o 10 input differenti cambia qualcosa? Si/No? Perché?
Chiunque proponga un test case deve saper rispondere a queste domande.

PERCHÉ ABBIAMO SCELTO UN CERTO TEST?
Il comportamento di un sistema viene analizzato basandosi sulle sue specifiche, che devono essere definite per ogni funzionalità del sistema da testare.
Secondo il **Boundary Testing**, in generale abbiamo tre categorie di casi di test:

- Casi scelti in quanto ***errori sicuri***.
- Casi scelti in quanto mostrano il ***normale funzionamento del sistema***.
- Casi ***bound*** *(estremi o di confine)*, ovvero i casi che sono tra l'errore e il corretto funzionamento *(es. ho un programma che accetta in input una sequenza di esattamente 6 cifre e ne inserisco 5 e/o 7)*.

SONO SUFFICIENTI I VARI CASI DI TEST PROPOSTI?
Per decidere se dei casi di test sono sufficientemente buoni per decidere se un programma è *(sufficientemente)* corretto, si utilizza un ***criterio di adeguatezza***, che non è altro che un set di test obligations.

Una test suite soddisfa un criterio di adeguatezza se:

- Tutti i test hanno successo *(passano)*.
- Ogni test obligation nel criterio è soddisfatto da almeno uno dei test nella test suite.

I criteri funzionali e strutturali sono ***complementari*** *(e quindi non mutualmente esclusivi)*.

Criteri funzionali: correttezza basata sulle specifiche del programma.
Esempio:

```
int fun(int param) {
    return param / 2;
}
```

Specifiche: Un programma che prende in input un intero N e restituisce $N/2$ se l'input è pari, altrimenti restituisce solo N.
È chiaro, osservando il codice, che questa implementazione funziona solo per i numeri pari; la ***failure*** quindi può rimanere nascosta se eseguiamo solo dei test strutturali, che si basano quindi solo sul codice.
Se eseguiamo dei **test funzionali** invece possiamo identificare la ***failure*** molto facilmente.
Considerando che il metodo in questione funziona bene dal punto di vista del codice, siamo di fronte ad un errore di ***missing logic***.

Generalmente si parte sempre dai test basati su criteri funzionali in quanto possono essere creati prima del termine dell'implementazione stesa del codice.

Criteri strutturali: correttezza basata sull'implementazione del programma.
Esempio:

```
void fun(int param) {
    if (param < 100) print(param * 2);
    else if (param < 1000) print(param);
    else print("too much!");
}
```

Specifiche: Un programma che prende in input un numero N e stampa $2N$ se N è minore 100, altrimenti stampa N.
Possiamo subito notare che l'implementazione non è corretta e che quindi c'è un ***fault***.
Un criterio funzionale non indica il bisogno di discriminare tra input <1000 e input $\geq$1000, ma questo è evidente nel codice, di conseguenza un **test strutturale** può identificare facilmente la ***failure***.

Inadeguatezza: cerchiamo di evitarla.
Esempi:

- *Funzionale*: se le specifiche descrivono un trattamento diverso in un set di casi, ma la test suite non controlla che quei determinati casi siano effettivamente trattati diversamente, possiamo concludere che la test suite è inadeguata ad evitare faults nella logica del programma.
- *Strutturale*: Se nessun test nella test suite esegue un particolare statement del programma, la test suite è inadeguata ad evitare faults nel codice.

In generale, se una test suite non soddisfa alcuni criteri, le obbligazioni non soddisfatte potrebbero fornire informazioni utili a migliorare la test suite, mentre se una test suite soddisfa tutte le obligations, ***non sappiamo definitivamente che è effettiva o meno***, ma abbiamo *qualche* prova di correttezza.

TESTARE CON 3, 5 O 10 INPUT DIFFERENTI CAMBIA QUALCOSA?
Iniziamo con l'introdurre due tipi di test: i test **Random** *(uniformi)* e i test **Sistematici** *(non uniformi)*;

RANDOM

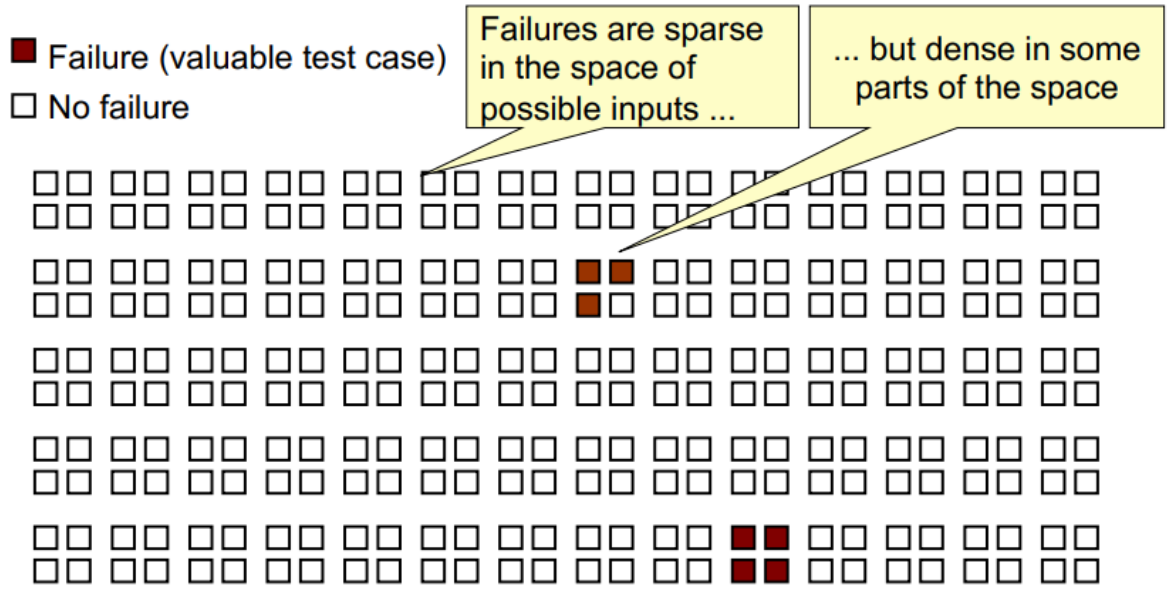
- Testa i possibili input uniformemente.
- Evita i pregiudizi del designer. Il test designer può tuttavia fare gli stessi errori logici e cattive assunzioni del designer del programma *(specialmente se sono la stessa persona)*.
- Associa a tutti gli input lo stesso valore.

SISTEMATICO

- Cerca di selezionare degli input che sono più *valuable* di altri.
- Solitamente lo fa selezionando dei *rappresentati* di alcune classi di input che tendono a fallire spesso o mai. I test funzionali sono un esempio di test sistematici.

Perché non random?
Perché nei software le faults non sono distribuite uniformemente; le ***failing values*** sono sparse nello spazio dei possibili input, di conseguenza è molto improbabile che un test random selezioni i valori che ci interessano per effettuare dei test.

Immaginiamo l'insieme dei possibili input come una serie di quadratini come mostrato sotto.



Assumendo che i **failure** siano densi in certi punti dello spazio di input, cerchiamo, secondo i principi del **test sistematico**, di **partizionare** tale insieme per definire delle classi che potrebbero o meno creare dei **failure**, così da avere un numero finito di elementi da testare.
Testando poi almeno un input nelle varie classi e trovando un **failure**, scopriamo il **fault**.

FUNCTIONAL TESTING

Un metodo per eseguire il partizionamento descritto sopra è il testing funzionale, conosciuto anche come **specification-based testing** e **black box testing**.
Il testing funzionale utilizza le specifiche del programma per definire queste partizioni dello spazio degli input per poi testare ogni categoria di input e i confini tra di esse *(**boundary testing**)*.

La specifica di un programma può essere lunga e confusionaria ed è per questo che è importante suddividerla in varie parti **testabili indipendentemente *(atomiche)*** ragionando su come l'applicazione si comporta in base a vari input, parametri o risultati. Questa parte di scomposizione viene effettuata con due metodi principali:

- **Catalog-Based testing** che aggrega e sintetizza l'esperienza degli altri test designers in un certo dominio, così da aiutare a trovare i valori *interessanti*; radunare esperienza in una collezione sistematica può velocizzare il processo di design, facendo diventare alcune decisioni una routine, automatizzandole per permettere di usare lo sforzo umano maggiormente sull'identificazione delle caratteristiche del sistema.
  In particolare, il metodo catalog based utilizza l'esperienza degli altri designers per elencare per determinate condizioni i casi più importanti dei possibili valori *(e.g. un catalog su una variabile intera potrebbe indicare di testate l'elemento sotto al limite inferiore/superiore, il limite stesso e un elemento nell'intervallo ammissibile)*.
- **Category-Partition testing** che identifica i test objectives e caratterizza l'input space.
  Lo fa scomponendo le specifiche in parti atomiche e identificando per ognuna di esse **parametri** e **variabili d'ambiente** *(file di configurazione, contenuti di database, ecc)*.
  Successivamente si concentra su ogni input/parametro/risultato per individuare delle caratteristiche *(categorie di valori)* che permettono di discriminare il comportamento del programma e applica su ognuna di queste il boundary testing.
  Eventualmente può anche introdurre dei limiti sulle possibili combinazioni di valori.

In seguito, avremo una fase di **aggregazione**, in modo da poter identificare gli input *interessanti*, fatta attraverso il **test combinatorio** che identifica i vari attributi che possono variare *(e.g. un browser potrebbe essere IE oppure Firefox, mentre un OS potrebbe essere Vista o XP)* e genera delle combinazioni che possono essere testate *(e.g. IE con Vista, IE con XP, Firefox con Vista, Firefox con XP)*.
Tuttavia, se abbiamo un grande numero di variabili in gioco, l'approccio combinatorio puro può essere sconveniente, in quanto produrrebbe troppe combinazioni da poterle testare tutte; usando il criterio **pairwise** invece eseguiamo delle combinazioni a coppia *(se il budget lo permette si possono usare anche triple, quadruple ecc)*, riducendo drasticamente il numero di combinazioni da testare senza introdurre dei limiti *(che però potremo introdurre se volessimo)*.

CONSTRAINTS
Per ridurre il numero di combinazioni testabili, possiamo anche introdurre dei limiti *(ad esempio per evitare di testare combinazioni impossibili)*.
I constraints si indicano con un'etichetta.

- **[error]**: indica un valore che corrisponde ad un errore e che quindi va testato una volta, ma difficilmente potrà essere combinato con altri valori.
- **[property][if-property]**: gestisce le combinazioni invalide di valori. In particolare **[property]** identifica un gruppo di valori con proprietà comuni, **[if-property]** limita la scelta di un valore che può essere combinato solo con un certo altro valore.
- **[single]**: indica un valore che l designer ***sceglie*** di testare una sola volta per ridurre il numero di test cases. Ha lo stesso effetto di *[error]*, ma è dettato da una scelta differente; è importante separarli per una corretta documentazione e per il regression testing.

SOMMARIO
Le specifiche dei requisiti generalmente sono espresse in linguaggio naturale, cosa che è un ostacolo per l'analisi automatica; il testing funzionale utilizza uno step manuale di **strutturazione** di specifiche in un set di **proprietà** e uno step automatizzabile di produzione delle **combinazioni testabili**.
Il metodo *brute force* è tedioso e porta ad errori; gli approcci combinatori decompongono il lavoro della forza bruta in diverse fasi per attaccare il problema in modo incrementale .

In particolare abbiamo ottenuto:

- Dal **Category Partition** la divisione del lavoro in uno step manuale di identificazione di categorie e proprietà e uno step automatico di generazione di combinazioni.
- Dal **Catalog Based** un miglioramento dello step manuale utilizzando dei pattern standard utili a identificare valori significanti.
- Dal **Pairwise Testing** la generazione sistematica di test suites più piccole.

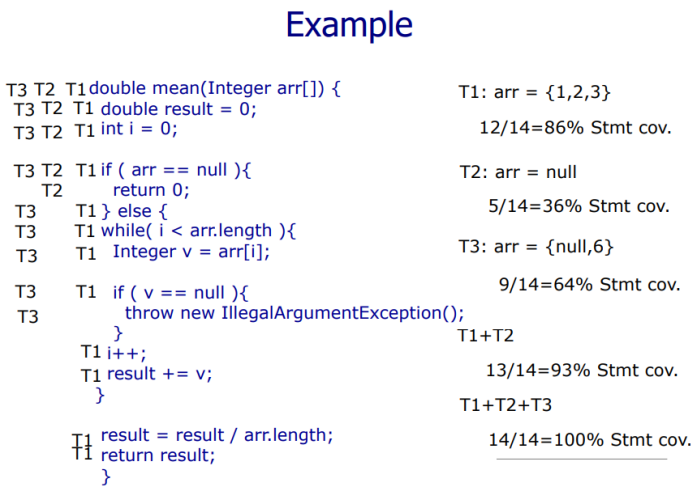
TEST STRUTTURALE

Il testing strutturale utlizza sia il software che la specifica e, in particolare, estende un insieme di test basandosi su entrambi; difatti le informazioni strutturali di un programma più che aiutarci a decidere come scegliere i test, risultano utili in combinazione con altri test *(funzionali)* per identificare ulteriori test atti ad identificare faults che potrebbero essere sfuggite all'analisi black-box.
Si può dire che il test strutturale è utilizzato per valutare la robustezza e la completezza delle test suite generate dai test funzionali.

Abbiamo visto che per decidere se una test suite è adeguata o meno possiamo usare i criteri di adeguatezza, ma come possiamo definire un criterio di adeguatezza per i test strutturali?
L'idea di base è di considerare ogni ***statement*** del programma come una test obligation e dire che l'adeguatezza di copertura degli statement è soddisfatta se ogni statement del programma è eseguito da almeno un test della test suite e il risultato dei test è *pass*.
Diremo che il **coverage** *(copertura)* di una test suite è dato dal #statement_eseguiti / #statement_totali.
Diremo inoltre che la test suite soddisfa il **criterio di adeguatezza** se questa ha coverage = 1 *(o a 100%)*.

In sostanza, diremo che il **criterio di copertura** è soddisfatto per un programma se testiamo tutte le sue istruzioni *(quindi ad esempio testare tutti i branch degli if)*.
Ovviamente la copertura **non** è direttamente proporzionale al numero di test della test suite, ma dipende più che altro da quante istruzioni eseguono effettivamente i test stessi. Inoltre spesso si preferisce un test rispetto ad un altro in basse alle informazioni che ci da sul tipo di fault rilevato e su dove questo si trovi.

Osserviamo l'esempio con tre test suite, che solo insieme eseguono tutti gli statement.



BASIC BLOCK TESTING

Un altro criterio di adeguatezza è il basic bloc testing, che considera dei blocchi di statement massimali che hanno un solo entry point ed un solo exit point *(ad esempio un **if** o un **while**)*. Sotto questa prospettiva, le istruzioni contigue appartengono ad uno stesso blocco finché non si incontra un *if*; gli statement condizionali terminano un blocco.
I cicli inoltre sono dei blocchi a se stanti.
L'idea alla base di questa strategia è quella di suddividere il programma in una serie di **basic blocks** da considerarsi ognuno come una ***test obligation*** e costruire un **Control Flow Graph *(CFG)*** che connetta i vari blocchi in base all'esito dello statement condizionale che termina il blocco *(ottenendo di fatto un DAG)*.

Il criterio di adeguatezza a questo punto è simile a quello dello **statement testing**, ovvero è soddisfatto solo quando la test suite esegue tutti i blocchi del CFG almeno una volta.
Difatti, il concetto alla base delle due strategie è lo stesso, la differenza sta nella granularità su cui si lavora: mentre nello statement testing testiamo le singole istruzioni, nel basic block testing testiamo una porzione di codice.

Tuttavia, nessuno dei due ci da alcuna garanzia sull'assenza di fault nel programma.
Consideriamo il seguente programma:

```
int abs(int x) {
    int r = 0;
    if (x < 0) {
        r = -x;
    }
    return r;
}
```

Consideriamo ora la test suite T_A = { -1 };
Notiamo che la test suite soddisfa i due criteri in quanto esegue tutti gli statement/blocchi del programma, tuttavia non individua il fault causato da un valore di x > 0 in cui il programma ritorna 0. Questo perché non stiamo considerando tutti i possibili esiti dello statement condizionale.

BRANCH TESTING

L'idea alla base del branch testing è che un programma ha tanti **branch** *(salti)*; ogni branch allora rappresenta una test obligation e deve essere testato almeno una volta. Come prima, la copertura risulta essere 1 quando la test suite testa effettivamente tutti i branch del programma.
Stiamo quindi praticamente creando un grafo proprio come con il basic block testing, in cui gli archi sono i branch del programma.
Visitando tutti gli archi sono quindi certo di visitare anche tutti i nodi; questo implica che se una test suite soddisfa il criterio del ***branch testing***, essa soddisfa anche il criterio dello ***statement/basick block testing*** *(mentre non è necessariamente vero il contrario)*.

BASIC CONDITION TESTING

Abbiamo visto fino ad ora una categoria di test strutturali basata sugli statements, che espone le faults legate a come una computazione è stata suddivisa i casi.
Tuttavia, perfino il criterio del branch testing può non individuare alcune faults.
Consideriamo il seguente programma che ritorna true a = -1 o b = -1:

```
boolean findMinusOne(int a, int b) {
    if (a == -1 || b == 1) {
        return true;
    } else {
        return false;
    }
}
```

Consideriamo ora la test suite T_R = { <2, 3>, <-1, 3> };
Notiamo che seppur essa soddisfi il criterio di adeguatezza del branch testing, non individua il fault **b == 1**.

Abbiamo bisogno quindi di un'altra categoria di test strutturali per individuare questo tipo di faults, i ***condition testing***, di cui il primo esempio è il ***basic condtion testing***.
Il criterio di adeguatezza del basic condition testing stabilisce che, per essere adeguata, una test suite deve eseguire almeno una volta ogni **condizione basica**. Il rapporto sarà quindi dato dal numero di valori di verità delle condizioni che abbiamo ottenuto e dal doppio delle condizioni stesse *(poiché ogni condizione può avere due outcome; T o F)*.
Consideriamo ora il programma di prima

```
boolean findMinusOne(int a, int b) {
    if (a == -1 || b == -1) {
        return true;
    } else {
        return false;
    }
}
```

e la seguente test suite T_C = { <2, -1>; <-1, 3> };
Notiamo che essa soddisfa il criterio di adeguatezza del **basic condition testing**, ma non esegue tutto il codice, in particolare non esegue il secondo **branch** del programma.
Possiamo quindi dedurre che il soddisfacimento del criterio del **basic condition testing *NON*** implica il soddisfacimento del criterio del **branch testing**.

BRANCH AND CONDITION

Una soluzione a questo problema è unire i due criteri.
Consideriamo le seguenti test suite T_A = { <-1, -1, -1>; <6, 7, 8> } e T_C = { <-1, 2, 3>; <2, -1, 2>; <6, 7, -1>; <6, 7, 8> }.
Notiamo che entrambe soddisfano i criteri di branch e condition, tuttavia T_C ci da più informazioni su dove si trovi un fault nel caso lo individuasse.

COMPOUND TESTING

Abbiamo detto che i criteri dei branch e delle basic conditions non sono comparabili in quanto nessuno dei due implica l'altro, dunque abbiamo bisogno di un test che controlli tutte le **combinazioni** di tutte le **condizioni** e percorra tutti i **blocchi di codice**; il ***compound testing*** effettua proprio questo lavoro.
Il criterio di adeguatezza del compound testing risulta soddisfatto solo se la test suite copre tutti i **branch** del programma E se valuta tutti i valori di tutte le **basic conditions**.

ATTENZIONE: poiché molti linguaggi di programmazione utilizzano ciò che è chiamata left-to-right/short-circuit evaluation, la quale implica che se abbiamo una catena di OR, non appena uno di essi risulta essere true, non andiamo a testare gli altri.
Discorso analogo vale per gli AND, ma con i false.
Questo ci permette di evitare di considerare tutte le possibili combinazioni in fase di testing.

Per esempio, per la condizione if(a == -1 || b == -1 || c == -1) basta la test suite
T = { <-1, *, *>; <2, -1, *>; <2, 3, -1>; <2, 3, 4>}.

Tuttavia, il compound testing ha un problema: il costo computazionale.
Difatti ha una **crescita esponenziale** dei casi di test da effettuare.
A volte la short-circuit evaluation riduce questo numero in modo che sia gestibile, ma non sempre.

MODIFIED CONDITION/DECISION

L'idea alla base del MC/DC è di effettuare i test solo per le combinazioni **importanti** di condizioni, in modo da evitare la crescite esponenziale.
Le combinazioni **importanti** sono quelle condizioni che, se cambiate **singolarmente**, portano ad un cambiamento dell'output.

WHY STRUCTURAL TESTING

Una possibile risposta è perché il test strutturale ci rivela cosa **manca** nella nostra test suite: se una parte di un programma non viene mai eseguita, le fault al suo interno non possono essere scoperte.

Il testing strutturale complementa quello funzionale; esso rappresenta un altro modo per individuare input trattati in modo diverso che però si affida al codice per farlo. Inoltre, il testing strutturale includere casi non scoperti nel testing funzionale, così come il testing funzionale può includere casi non scoperti nel test strutturale.

Tuttavia, eseguire tutti gli elementi del control flow **NON GARANTISCE** l'assenza di fault nel codice, il testing strutturale rimuove le inadeguatezze ovvie, ma non da alcuna garanzia di trovare tutti i fault.

A volte, nessuna test suite può soddisfare un criterio per un programma dato; in questo caso si possono escludere gli statement non raggiungibili *(tuttavia non possiamo sempre sapere per certo quali statement non verranno mai eseguiti)* oppure misurare quanto copre del programma la test suite.

SECURITY IS NOT SAFETY

Definiamo:

- **SAFETY ENGINEERING**: prevenzione di danni *'catastrofici'* causati dal cattivo funzionamento di un sistema informatico.
- **SECURITY ENGINEERING**: prevenzione, difesa e recupero danni contro la possibilità di attacchi che possono mettere a repentaglio i valori rappresentati da un sistema informatico.

Tutte le misure di **safety** sono anche misure di **security**, mentre non è vero il viceversa.

Un problema di sicurezza, o *vulnerabilità*, può essere viso come una ***via alternativa***, un modo di usare il sistema non pensato dal progettista. Ovviamente più è complesso il sistema, più possono essere le vie alternative.
Esempio: un progettista di software di un auto potrebbe pensare che sia utile che le portiere si sblocchino quando la macchina si capovolge. Tuttavia, una via alternativa potrebbe essere far pressione sul tetto per aprire la macchina quando dovrebbe rimanere chiusa.

CIA TRIAD

- **Confidenzialità**: fa riferimento al fatto che una risorsa *(o un servizio)* deve essere accessibile solo a chi è autorizzato ad usufruirne.

  Può essere descritto anche come capacità di controllare/assicurare la **privatezza**/**riservatezza** delle informazioni rispetto a soggetti non autorizzati.
  Questo concetto non si applica solo, per esempio, al contenuto di un file, ma anche alla conoscenza dell'esistenza di tale file o meno o anche all'utilizzo di un certo mezzo *(ad esempio la rete)*.

- **Integrità**: fa riferimento al concetto di assicurazione che un'informazione non sia stata modificata e che quindi la risorsa sia fruibile nel modo esatto nel quale è stata resa disponibile.

  Può essere descritto anche come capacità di controllare/assicurare la **non modificabilità**/**attendibilità** da parte di soggetti non autorizzati delle informazioni.
  Il concetto di modificabilità si estende anche alla creazione/rimozione di oggetti *(anche se nel caso di rimozione si parla anche di interruzione di disponibilità)*, oltre che alla manomissione di informazioni.

  Due particolari sotto-casi del principio di integrità sono:

  1. **Autenticazione di mittente**: definisce la capacità di garantire l'identità del mittente.
  2. **Non-ripudio**: definisce la capacità di assicurare che il mittente non possa negare la paternità delle informazioni trasmesse.

- **Disponibilità** *(Availability)*: fa riferimento al fatto che se una risorsa deve essere accessibile a degli utenti, un attacco non deve inibire questo accesso *(esempio un attacco DDoS)*.

  Può essere descritto anche come capacità di controllare/assicurare la **possibilità di usare** risorse e servizi da parte degli utenti autorizzati.
  Alcuni esempi di attacchi alla disponibilità sono il crash di un sistema, la manomissione di algoritmi di scheduling per non far mai eseguire un certo processo o anche la distruzione fisica di un computer.

Gli attacchi alla triade CIA sono solitamente identificati come *DAD*:

- **Disclosure** > Confidentiality
- **Alteration** > Integrity
- **Destruction** > Availability

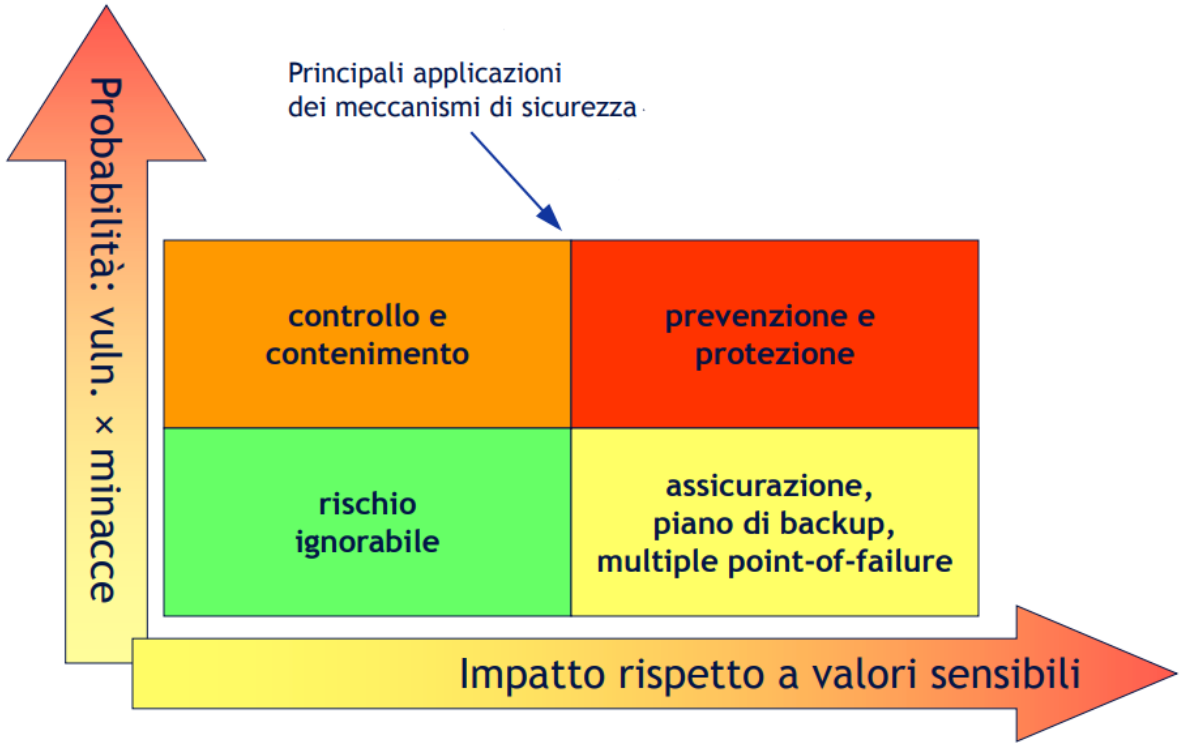
USABILITY VS SECURITY
L'obiettivo di fondo dell'informatica è favorire l'accesso alle risorse, mentre la sicurezza obbliga a limitarne l'accesso. Dunque spesso l'obiettivo della sicurezza è trovare il giusto compromesso tra usabilità e sicurezza del sistema.

Inoltre, spesso la sicurezza viene presa in considerazione solo dopo i primi incidenti; in realtà il compromesso citato sopra andrebbe preso in considerazione fin da subito, trovando un compromesso anche tra benefici e costi, che quindi richiede un'*analisi dei rischi*.

ANALISI DEI RISCHI
Rischio: **possibilità** di **perdita** o danno.
Nell'ambito della sicurezza il rischio è calcolabile come **Rischio = Vulnerabilità x Minacce x Valori** dove:

- **VALORI**: solo le parti del sistema informativo importanti, sensibili, critiche, da proteggere.
- **VULNERABILITÀ**: debolezze del sistema, attacchi che possono essere sfruttati per comprometterlo.
- **MINACCE**: circostanze o agenti che possono/vogliono arrecare danni.

Notiamo il fattore moltiplicativo; se le vulnerabilità sono poche, il rischio è basso, viceversa se le minacce sono basse si corrono pochi rischi. Se i valori non sono importanti, anche se ci fosse un attacco la perdita è minima.

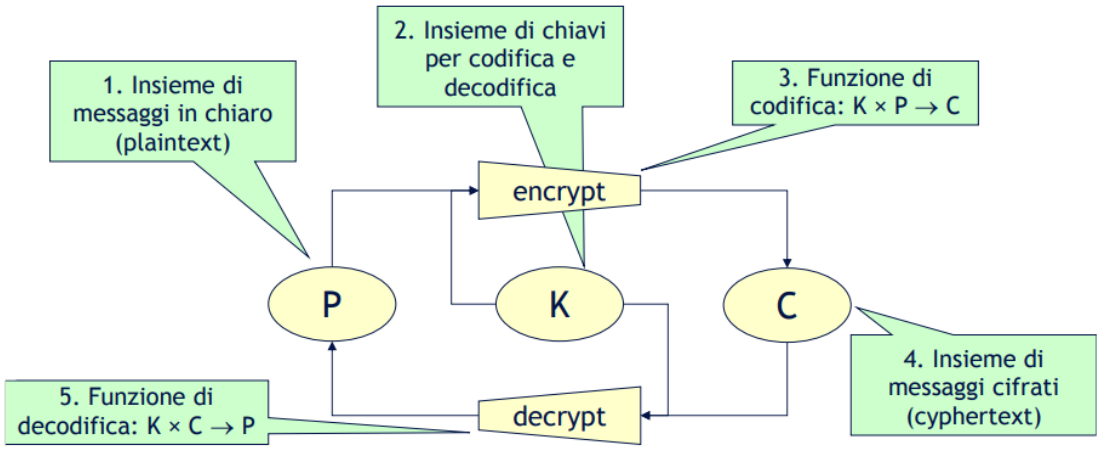


CRITTOGRAFIA

La crittografia *(dal greco **kryptos**, nascosto)* è una disciplina che studia tecniche per *codificare/decodificare* messaggi in modo da garantire la privacy della comunicazione fra due soggetti.
Essa **NON** è la soluzione per tutti i problemi di sicurezza, ma può essere uno strumento importante se implementata in modo corretto.

Generalmente un sistema crittografico fa uso di un algoritmo di crittografia governato da una **chiave** che permetta di garantire la confidenzialità ed integrità del messaggio e che permetta al destinatario di poterlo leggere.

Un sistema crittografico è formato da 5 parti



Un primo esempio di sistema crittografico è il **cifrario di Cesare** che consiste nello spostamento delle lettere utilizzando l'aritmetica in modulo n. Avremmo quindi:

- **P**: lettere alfabeto
- **C**: lettere alfabeto
- **K**: numeri da 1 a 25
- **encrypt(k, p)**: carattere a distanza k nell'alfabeto
- **decrypt(k, c)**: carattere a distanza -k nell'alfabeto

Il principale problema di questo sistema è il basso numero di chiavi disponibili, che lo rende molto vulnerabile ad un attacco di forza bruta.

CRITTOGRAFIA E CRITTOANALISI

- **Crittografia**: schemi/metodi/algoritmi per la codifica sicura dei messaggi.
- **Crittoanalisi**: rappresenta il tentativo di *violare* la crittografia attraverso lo studio di insiemi di messaggi crittati. È utile per scoprire le debolezze di un algoritmo crittografico o del suo ambiente di funzionamento.
  Si pone come obiettivo non solo quello di scoprire un messaggio, ma anche di scoprire le chiavi usate per la cifratura o addirittura dedurre significati segreti senza necessariamente decifrare i messaggi *(ad esempio guardando la frequenza di invio dei messaggi o la loro lunghezza)*.

Un algoritmo crittografico è violabile se può essere violato da un crittoanalista **a patto di disporre di risorse e tempo sufficienti** *(se non consideriamo il tempo e le risorse, teoricamente, ogni algoritmo è violabile con forza bruta)*.

Gli algoritmi ***degni di fiducia*** solitamente sono basati sulla matematica, analizzati da esperti e in uso da diverso tempo senza essere violati.
Ma devono anche essere **usabili**, ovvero il tempo necessario per crittare/decrittare i messaggi deve essere commisurato alla sicurezza necessaria, le chiavi non devono essere soggetti a requisiti complessi e il processo crittografico non deve essere troppo complesso per l'utente.

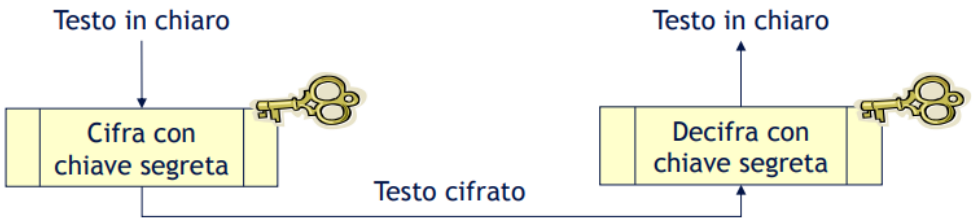
Ovviamente la prima cosa che pensiamo alla crittografia è la **confidenzialità** dei messaggi, ma gli algoritmi crittografici vengono usati anche per garantire **integrità** e **non ripudio**, similmente a quanto accade con i documenti fisici.

Gli algoritmi crittografici si dividono in 3 classi:

- **Algoritmi simmetrici a chiave segreta**: viene usata un'unica chiave sia per crittare che decrittare.
- **Algoritmi asimmetrici a chiave pubblica**: vengono usate chiavi diverse per crittare e decrittare.
- **Algoritmi di hashing**: il messaggio stesso è la chiave e servono a garantirne l'integrità.

SISTEMI SIMMETRICI

Abbiamo detto che gli algoritmi simmetrici usano una stessa chiave per decrittare e crittare, dunque decrypt(k, encrypt(k, p)) = p.
Dunque, per istanziare una comunicazione confidenziale, due soggetti devono conoscere una chiave *k* non nota a nessun altro.

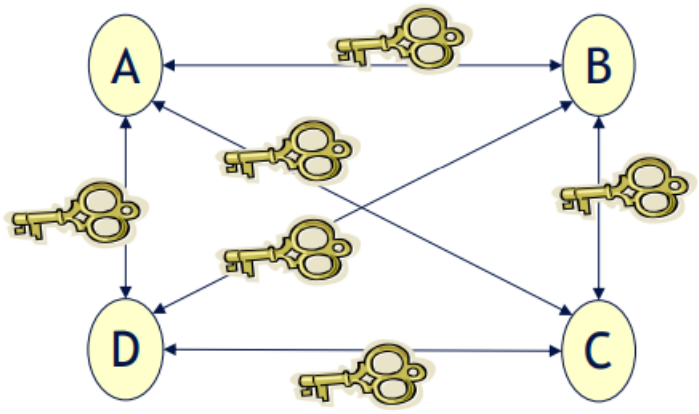


Assumendo che mittente e ricevente siano gli unici a conoscenza della chiave:

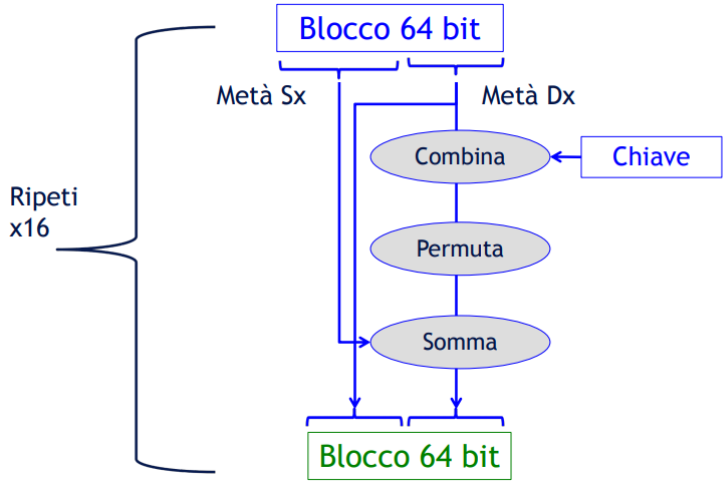
REQUISITO	SI/NO	MOTIVO
Confidenzialità	Si	Solo chi conosce la chiave segreta può decodificare il messaggio.
Integrità	Si	Una volta crittato, il messaggio non è modificabile prima della decrittazione. Se venisse modificato il messaggio crittato, questo verrebbe corrotto.
Autenticazione e non ripudio	Si	Solo chi conosce la chiave segreta può crittare e quindi essere mittente del messaggio *(assumendo che il sistema funzioni e che quindi la chiave non venga condivisa)*.

In questo caso **non è possibile** garantire uno solo di questi requisiti indipendentemente dagli altri.

Un primo aspetto *controverso* di questi algoritmi riguarda lo scambio della chiave che deve avvenire ovviamente in modo sicuro, ma ne parleremo successivamente.
Un altro aspetto di questo tipo di algoritmi è il numero di chiavi necessarie; per N individui che comunicano in maniera sicura a coppie abbiamo bisogno di $n(n-1)/2$ chiavi *(una per coppia, un numero quadratico)*!



DES
Un tipico algoritmo simmetrico è **DES: Data Encryption Standard** che codifica i messaggi in blocchi da 64 bit e che consiste nell'applicare iterativamente per 16 volte una funzione combinatoria ad ogni blocco usando una chiave di 56 bit come uno dei parametri della funzione stessa.



Ovviamente, essendo un algoritmo simmetrico, lo stesso algoritmo che viene usato per crittare i messaggi viene usato anche per decrittarli. Inoltre, la particolarità di DES è che opera sui **bit**: combina i bit, permuta i bit, somma i bit. Questo rende l'algoritmo particolarmente adatto all'**implementazione su hardware** e quindi è **molto veloce** come algoritmo.

Alla sua uscita, nel 77, DES risultava non violabile con forza bruta *(ci sarebbero voluti 700 anni con la potenza computazionale di allora)*, tuttavia nel 97 venne risolto in circa 4 mesi usando la computazione parallela. Con la potenza computazionale di oggi probabilmente ci vuole poco a violare tale algoritmo.

Come risposta a ciò è stato creato triple DES che cifra 3 volte con 3 chiavi diverse, aumentando notevolmente il tempo stimabile per un attacco di forza bruta.

Un'evoluzione di DES è **AES (Advanced Encryption Standard)** che si basa sulle stesse operazioni di DES ma è estendibile, ovvero permette di usare chiavi di lunghezza variabile *(solitamente vengono usate chiavi da 128, 192 o 256 bit ma si può andare oltre)* e permette di eseguire più cicli di codifica, rendendolo di fatto molto più sicuro.